CS422 - Programming Language Design

Lecture 10: Typed Languages - Static Type Checking

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

There was little in the definition of our functional programming language so far which would forbid one from writing erroneous programs. While the domain or application specific errors are hard or impossible to catch automatically, there is a large class of errors, known as *type errors*, which often can be caught automatically with appropriate program analysis tools.

Static type checkers are special program analysis tools that take programs as inputs and report type errors, which may turn into runtime errors or misbehaviors. Type checkers are so popular nowadays and the kind of errors they catch so frequently lead to wrong behaviors, that most compilers today provide a static type checking analysis tool as a front end. The major drawback of static type checkers is that they may *reject correct programs*, so they somehow limit the possibilities of writing programs.

Typed Languages

A typed language defines a set of *types*, as well as a *typing policy* by which types can be associated to values calculated by programs.

By applying the typing policy recursively, one can mechanically associate a type to each expression in a given program. In order to do it, one needs to inspect the invocations of language constructs and check that each language construct is used properly by analyzing the relationship between its operands, their expected types and the context in which the language construct appears.

If a language construct use is found to violate the expected typing policies, then we say that a *type error* has been found. E.g., think of the expression `3 + (fun x -> x)`. While under certain special circumstances one may wish to accept such programs as correct, in general a typing policy would classify this is as a type error.

When designing a type checking tool for a typed language, one needs to take a decision on *what to do when a type error is found*.

The simplest decision is to reject the program. Better decisions are to also report where the error has been found, as well as possible causes, to users. Even better, the type checker can continue and report subsequent type errors within only one session.

One can even think of taking automatically some correcting measures, such as to convert integers to reals, or kilometers to miles, or to modify the code by introducing runtime checks, etc.

Dynamic Type Checking

At the expense of increasing their runtime overhead, some languages maintain a type together with each computed value in their store. This way, variables are associated not only values but also the types of those values. Before an operation is applied, a *dynamic type checker* validates the application of the operation. In the case of `3 + (fun x -> x)`, before the operation `_+_` is evaluated the dynamic type checker checks whether the two operands are integers; this way a runtime error will be reported.

While dynamic type checking is useful and precise, and many successful programming languages are dynamically typed, such as *Scheme*, *PERL*, *Python*, etc., most programming language designers believe that runtime checks are too expensive in practice.

Untyped Memory Models

If one simply removes the runtime checks for type consistency then one can obviously run into serious difficulties. For example, when adding an integer and a function, since they are both stored as binary numbers at some locations, without a consistency check one would just add the integer with the binary representation of (part of) a particular representation of the function, which is wrong.

Languages without runtime or dynamic type checking are said to have *untyped memory models*. Such languages essentially assume that the operations are applied on the right data, so they can increase their performance by removing the runtime type consistency checks.

Static Type Checking

To take full advantage of the increased execution speed of untyped memory models while still not sacrificing the correct runtime behavior of programs, an additional layer of certification is needed. More precisely, one needs to ensure *before execution* that all the operations will be applied on the expected type of data. This process is known under the terminology *static type checking*.

Languages admitting static type checkers, that ensure that once a program passes the type checker it will never exhibit any type error at runtime, are called *strongly statically typed*. It is quite hard or even impossible (due to undecidability arguments) in practice to devise strongly typed languages where the types are intended to state that a program is free of general purpose errors. For example, it is known to be undecidable to say whether a program will ever perform a division by zero, or if a program terminates.

To keep static type checkers decidable and tractable, one typically has to *restrict* the class of errors that types can exhibit. In the context of our simple functional programming language, like in many other languages, by a *type error* we mean one of the following:

- An application of a non-functional expression to an argument;
- An application of a function expression to arguments whose types are different from those of function's parameters;
- An assignment of an expression to an existing name, such that the type of the expression is different from the declared type of the name;
- A use of an arithmetic or boolean operation on non-integers or non-booleans, respectively;
- A use of a conditional statement where the first argument is not a boolean or where the two branches are of different type.

Besides the simplistic typing policy above, we also need to state *how the types of expressions are calculated and propagated* via *typing rules*. This is quite straightforward in our formalisms, because typing rules are nothing but special cases of *conditional equations*. For example, "the type of `x + y` is `int` if the types of `x` and `y` are `int`". Similarly, "the type of a conditional is the type of its first branch if the type of its argument is `bool` and the types of its branches coincide".

In order to allow type checking in a language, one first needs to extend the language with *type declarations*, so that one can add typing information to the program. We will see later, when we discuss type inference, that in some situations types can be deduced automatically by examining how names and expressions are used.

Since declarations can occur either in a `let`/`letrec` statement or in a function (its parameter), we will slightly extend the syntax of our language to allow type declarations in addition to and at the same

time with name declarations. For example, we will allow expressions like

```
let int x = 17 in x
let int x = 17 and int y = 10 in x + y
```

Besides the *basic types* `int` and `bool`, we will also introduce several other types and type constructors. Since our language is functional we will need to introduce *function types*, using the common type constructor `_->_`. Thus, we will be able to write and type-check expressions like

```
let int a = 3
in let (int -> int) p = fun(int x) -> x + a
   and int a = 5
   in a * p(2)
```

Note that, by design, we decided to first place the type of a parameter and then its name. As before, we also assume that the

default calling mode is call-by-value. So expressions like the one below will also type check (note also the tricky typing of `letrec`):

```
letrec int x = 2
and    (int * int -> int) f = fun (int y, int z) -> y + x * z
in f(1,x)
```

Defining Program Analysis Tools

The modular design of our programming language was intended not only to easily deal with changes in the design of the language, but also to facilitate the definition of *program analysis tools*.

In fact, the definition of a program analysis tool can be seen as very similar in spirit to defining a semantics to a programming language. The meaning a program analysis tool has for a program may, of course, be quite different from that of the semantics of the

programming language, because the two may look at two quite different aspects of a program.

However, we will again take advantage of the efficient executable environment provided by Maude through its fast implementation of rewriting, and this way obtain not only executable but also quite efficient program analysis *tools for free*, from just their mathematical rigorous definition.

In today's lecture we will focus on one of the simplest possible program analysis tools, namely a static type checker.

Adding Types to the Syntax

There are very few changes that need to be done in the syntax of our functional programming language in order to allow type declarations. First, of course, one has to introduce the types:

```
fmod TYPE is
  sort Type .
  ops int bool unit none : -> Type .
  op _->_ : Type Type -> Type [gather(e E)] .
  op list_ : Type -> Type [prec 0] .
endfm
```

Besides `bool` and `int`, note that there are two more basic types: `unit` and `none`. The former will be the type of `()` and the later will be the type of expressions that are supposed to evaluate to `nothing` (assignments, loops, etc.). The type of a function without parameters returning an integer will be therefore `unit -> int`. To

avoid writing parentheses, note that the function type constructor, _->_ has been defined as right-associative. Note that this is opposite to the definition of function application, which was defined left-associative, and that these two conventions are consistent.

Parameters now need to be typed, so their syntax needs to also be slightly changed. The module below shows the syntax of the typed parameters. The major conceptual change is that each name comes now with a type, the rest being just beautification of the spec:

```
fmod PARAMETER-PASSING-SYNTAX is protecting NAME .  protecting TYPE .
  sorts Unit Parameter ParameterSeq PassingMode .
  subsorts Unit < Parameter < ParameterSeq .
  op `(`) : -> Unit .
  op ___ : Type PassingMode Name -> Parameter [prec 0] .
  op __ : Type Name -> Parameter [prec 0] .
  op __ : ParameterSeq ParameterSeq -> ParameterSeq [gather(E e) prec 1]
endfm
```

Exercise 1 *We could have designed the syntax of our programming language in a more modular way such that one could have replaced all the changes to syntax above by adding just one prefix "attribute" to a properly defined "generic" name. The calling mode and the type, and perhaps other annotations needed in the future, would be just other attributes of such a generic name. Do this change in the design of our language and then reflect on the increase in modularity attained this way.*

The complete syntax of the typed `FUN` can be found in the provided Maude file `fun-type-checking-syntax.maude`.

Defining a Static Type Checker

The general idea underlying a static type checker is to recursively analyze each language construct occurring in a given program, to prove that its operands satisfy the type constraints, and then to assign a type to the expression created by that new construct.

The type of a name cannot be changed by variable assignment constructs, so *a name has its declared type in all its occurrences within its scope*; but clearly one can differentiate between static and dynamic scoping. We will only consider static scoping.

Exercise 2 (Very hard!). *Define an static type checker for the dynamically scoped version of* `FUN`*. Can the task be accomplished precisely? If not, choose some acceptable trade-offs so that the type checker stays reasonably efficient.*

Defining the State Infrastructure

Any software analysis tool has a state that it maintains as it traverses the program to be analyzed. This is also what happened in the definition of the semantics of the language. In fact, there is a striking similarity in giving various executable semantics to a given syntax: a programming language semantics is just one possibility; a type checker is another one; other software analysis tools can be defined similarly, including complex type systems or abstractions.

A type checker ignores the concrete values bound to names, but it needs instead to *bind names to their types*. We can realize that by defining a special state attribute, called `TypeEnv`, which contains a mapping from names to their current types. In order to do define this module, we need to first define lists of names and of types. The former were already defined in the semantics of `FUN`, and the latter are similar, so we do not discuss them here. See the provided file

`fun-type-checking-semantics.maude` for the complete Maude specification of the static type checker discussed in this lecture.

The following module can simply "cut-and-paste" the module `ENVIRONMENT` defined by the executable semantics of `FUN`; however, one needs to replace locations by types. The current version of Maude does not provide support for parameterized modules, like OBJ. If it did, then we could have defined a generic environment module as a map parameterized by its domain and target sorts, and then just instantiate it whenever needed.

```
fmod TYPE-ENVIRONMENT is
  protecting NAME-LIST .
  protecting TYPE-LIST .
  sort TypeEnv .
  op empty : -> TypeEnv .
  op [_,_] : Name Type -> TypeEnv .
  op __ : TypeEnv TypeEnv -> TypeEnv [assoc comm id: empty] .
  op _[_] : TypeEnv Name -> [Type] .
  op _[_<-_] : TypeEnv NameList TypeList -> TypeEnv .
  var X : Name .  vars TEnv : TypeEnv .  vars T T' : Type .
  var Xl : NameList .  var Tl : TypeList .
  eq ([X,T] TEnv)[X] = T .
  eq TEnv[nil <- nil] = TEnv .
  eq ([X,T] TEnv)[X,Xl <- T',Tl] = ([X,T'] TEnv)[Xl <- Tl] .
  eq TEnv[X,Xl <- T,Tl] = (TEnv [X,T])[Xl <- Tl] [owise] .
endfm
```

In this case, no other information but the type environment is needed by the static type checker. However, to preserve our general semantic definition methodology, we prefer to define a state of the type checker which contains only one state attribute for the time being, the type environment. Extensions of the language may require new state attributes in the future:

```
fmod TYPE-STATE is
  extending TYPE-ENVIRONMENT .
  sorts TypeStateAttribute TypeState .
  subsort TypeStateAttribute < TypeState .
  op empty : -> TypeState .
  op __ : TypeState TypeState -> TypeState [assoc comm id: empty] .
  op tenv : TypeEnv -> TypeStateAttribute .
endfm
```

We can now finalize the state infrastructure by defining the following module, which, like for the semantics of language,

provides an abstract state interface for the subsequent modules:

```
fmod HELPING-OPERATIONS is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE-STATE .
  sort ExpList .
  subsorts Exp NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  op initialTypeState : -> TypeState .
  op _[_] : TypeState Name -> [Type] .
  op _[_<-_] : TypeState NameList TypeList -> TypeState .
  var S : TypeState .  var TEnv : TypeEnv .
  var X : Name .  var Xl : NameList .  var Tl : TypeList .
  eq initialTypeState = tenv(empty) .
  eq (tenv(TEnv) S)[X] = TEnv[X] .
  eq (tenv(TEnv) S)[Xl <- Tl] = tenv(TEnv[Xl <- Tl]) S .
endfm
```

`S[X]` gives the type associated to a name `X` in a state `S`, while `S[Xl <- Tl]` updates the types of the names in the list `Xl` to `Tl`.

Typing Generic Expressions

Like in the case of the semantic definition of `FUN`, we need to "evaluate" an expression to a "value". However, this time, the value that expressions will be evaluated to will be types. Unlike in the definition of the semantics of the language where side effects could modify the values bound to names, *the type bound to a name cannot be modified*.

Therefore, we only need one operation, say `type`, returning the type of an expression; there is no need to propagate states and "side effects". Since we eventually need to handle lists of names, expressions and types, because of `let` and `let rec`, we prefer to define it directly on lists of expressions. Note that the result of this operation is a bracketed sort, or a kind in Maude terminology, reflecting the fact that some expressions may now be typed. We will see many such expressions in the sequel.

```
fmod GENERIC-EXP-STATIC-TYPING is extending HELPING-OPERATIONS .
  vars E E' : Exp .  var El : ExpList .
  var S : TypeState .  var I : Int .  var X : Name .
  op type : ExpList TypeState -> [TypeList] .
  eq type((E,E',El), S) = type(E, S), type((E',El), S) .
  eq type(I, S) = int .
  eq type(X, S) = S[X] .
endfm
```

Typing Arithmetic and Boolean Expressions

To keep the semantics of the type checker compact, we will use conditional equations in what follows. However, as in the case of the (first) semantics of FUN, conditional equations are not necessary. A homework exercise will ask you to provide an unconditional semantics.

We can now define the *typing rules* for the arithmetic and boolean operators simply as their corresponding conditional equations. For example, the type of `E + E'` in a state `S` is `int` if both `E` and `E'` have the type `int` in `S`:

```
...
 ceq type(E + E', S) = int if type((E,E'), S) = (int, int) .
...
```

The typing of boolean expressions can be defined similarly:

```
...
 ceq type(E eq E', S) = bool if type((E,E'), S) = (int, int) .
 ceq type(E and E', S) = bool if type((E,E'), S) = (bool, bool) .
 ceq type(not E, S) = bool if type(E, S) = bool .
...
```

Note that being able to type a list of expressions to a list of types, as opposed to typing just one expressions, helps us write more compact and readable definitions.

Typing the Conditional

The typing policy of `if_then_else_` states that its first argument should be of type `bool`, while the other two arguments should have the same type. This policy can be simply stated as follows:

```
...
 ceq type(if BE then E else E', S) = T
    if (bool,T,T) := type((BE,E,E'), S) .
...
```

Note that `T` occurs twice in the pattern in the condition. This is perfectly correct; remember that the same matching algorithm used for the left-hand-sides of equations is used once the matched term is reduced to its normal form.

Typing Functions

Functions and their applications are, in some sense, the trickiest to type. As in the case of the semantics of FUN, we first apply currying and then only need to worry about defining the typing semantics of one parameter functions. An important observation in the context of typing, is that the *parameter passing style does not play any role*, so we can ignore the passing modes (Pm has sort PassingMode):

```
...
  eq T Pm X = T X .
```

Function declarations can now be typed as follows:

```
  eq type(fun T X -> E, S) = T -> type(E, S[X <- T]) .
  eq type(fun () -> E, S) = unit -> type(E, S) .
```

So the body is typed in the declaration environment updated with a binding of the function's parameter, if any, to its declared type.

The type of `()` is `unit`, which was already reflected by the last equation above. However, `()` can also occur as an ordinary expression, so it needs to also be typed in isolation:

```
eq type((), S) = unit .
```

Finally, the type of a function application can be defined now. The typing policy of a function invocation `F E` is as follows: `F` should first type to a function type, say `Tp -> T`; then `E` should type to the same type `Tp` that occurred in the type of `F`; if this is the case, then the type of the function application is, as expected, `T`. All these words can be replaced by a straightforward conditional equation (note, again, the double occurrence of a variable in the pattern):

```
ceq type(F E, S) = T if ((Tp -> T), Tp) := type((F,E), S) .
```

Typed Bindings

Since the names bound in `let` and `let rec` constructs are now typed, we need to change our previous grouping operator (in `FUN`'s semantics), that grouped the bound names into one list and the corresponding binding expressions into another list, to also group the corresponding types. The changes below are straightforward:

```
...
  op `(_,_,_`) : TypeList NameList ExpList -> Bindings .
...
  eq (T X = E) = (T, X, E) .
  eq (Tl, Xl, El) and (Tl', Xl', El') = ((Tl,Tl'), (Xl,Xl'), (El,El')) .
...
```

Typing `let`

The type of a `let` expression is the type of its body in the state obtained after binding its names to the types of the corresponding expressions:

```
ceq type(let (Tl,Xl,El) in E, S) = type(E, S[Xl <- Tl])
   if Tl = type(El, S) .
```

The type of a `let` remains undefined if its bindings cannot be properly typed. This is assured by the fact that `Tl` is declared as a variable of sort `TypeList`, so `type(El, S)` must reduce to a proper list of types in order for the equation to be applied.

Typing `let rec`

Typing of `let rec` is relatively easy once we have the current infrastructure. In order to find its type, one

- first blindly binds all its names to their declared types;
- then checks the type consistency of the bound expressions to their corresponding names' types;
- then, if each of those are correctly typed, returns the type of its body expression in the new state:

```
ceq type(let rec (Tl,Xl,El) in E, S) = type(E, S[Xl <- Tl])
   if Tl = type(El, S[Xl <- Tl]) .
```

Typing Lists

The typing policy of lists is that each element in the list should have the same type, say `T`; if this is the case, then the type of the list expression is `list T`. For example, `[1 :: 2 :: 3]` is typed to `list int`, while `[1 :: true :: 3]` generates a type error.

Hence, we need an operator that takes an `Exp::List` and returns the type of its elements in case there is one. We call it also `type`:

```
op type : Exp::List -> [Type] .
ceq type(E :: E::l, S) = T if T := type(E, S) /\ T = type(E::l, S) .
```

The type of a list can be simply defined as follows:

```
eq type([E::l], S) = list type(E::l, S) .
```

How about the type of the empty list, `[]`? It must be a list of some type, but what type? This is a subtle problem which cannot be

solved easily. It is related to the capability of many functional languages to allow *polymorphism*, that is, to allow expressions to have parametric types. We will discuss polymorphism next lecture in the context of *type inference*, when we will also see the clean solution to the typing of an empty list. For now, to keep our language type-checkable, let us make the strong and admittedly unsatisfactory assumption that all empty lists have type `list int`:

```
eq type([], S) = list int .
```

The typing of the list operations is self-explanatory:

```
ceq type(car(E), S) = T if list T := type(E, S) .
ceq type(cdr(E), S) = list T if list T := type(E, S) .
ceq type(cons(E,E'), S) = list T if (T, list T) := type((E,E'),S) .
ceq type(null?(E), S) = bool if list T := type(E, S) .
```

Typing Assignment and Sequential Composition

A variable assignment statement should not modify the type of the name on which it applies its side effect. Also, since we decided to evaluate assignment statements to the value `nothing`, we also type them to the special type `none`, so the type system will signal if one uses them in wrong contexts, for example as integers:

```
ceq type(X := E, S) = none if type(X, S) = type(E, S) .
```

The type of `E ; E'` is the type of `E'`, but only if `E` also can be correctly typed, say to some type `T`:

```
ceq type(E ; E', S) = T' if (T, T') := type((E,E'), S) .
```

Typing Blocks and Loops

The empty block {} was evaluated to `nothing` in the semantics of `FUN`, while a block {`E`} was evaluated to the value of `E`. This suggests the following typing of blocks:

```
eq type({}, S) = none .
eq type({E}, S) = type(E, S) .
```

Note that one is free to choose a different typing policy, such as, for example, one stating that the type of a block is always `none`. We choose a similar policy for loops:

```
ceq type(while Cond Body, S) = none
   if (bool,T) := type((Cond, Body), S) .
```

Note that the type of a loop is `none` only if its condition types to `bool` and its body to some proper type. We could have just as well required that the type of its body be `none`, or that the type of the loop is the type of its body, etc.

Putting the Type Checker Together

Like in the semantics of FUN, we can now put everything together by defining one unifying module importing all the various features. This module also defines an operation that can check and calculate the type of an expression in the initial state:

```
op type : Exp -> [Type] .
eq type(E) = type(E, initialTypeState) .
```

Homework Exercise 1 *The executable semantics of the type checker above used* conditional equations*. While this is not a problem in the context of Maude, it may be a serious limitation in the context of other equational engines that do not provide support for matching in conditions or even for conditional equations of any kind. Give an* unconditional equational *semantics, also using Maude, to the type checking tool defined in this lecture.*